Under consideration for publication in J. Functional Programming

Effekt

Type- and Effect-Safe, Extensible Effect Handlers in Scala

JONATHAN IMMANUEL BRACHTHÄUSER, PHILIPP SCHUSTER, and KLAUS OSTERMANN
University of Tübingen, Germany
(*e-mail:* {jonathan.brachthaeuser,philipp.schuster,klaus.ostermann}@uni-tuebingen.de)

Abstract

Effect handlers are a promising way to structure effectful programs in a modular way. We present the Scala library Effekt, which is centered around capability passing and implemented in terms of a monad for multi-prompt delimited continuations. Effekt is the first practical implementation of effect handlers that supports effect safety, effect polymorphism, effect parametricity, and effect encapsulation which means that all effects are handled and effects cannot be accidentally handled by the wrong handler. Other existing languages and libraries break effect encapsulation by leaking implementation details in the effect type unless the user manually adds lifting annotations. We describe a novel way of achieving effect-safety using intersection types and path dependent types. We represent effect rows as the contravariant intersection of effect types, where an individual effect is represented by the singleton type of its capability. Handlers remove components of the intersection type. By reusing the existing type system we get subtyping and polymorphism of effects for free. The effect system not only guarantees safety, but also guarantees modular reasoning about higher-order effectful programs.

1. Introduction

Consider the following piece of code written in Scala and using our library Effekt. The program uses two *effect operations* `flip` for nondeterministic coin flipping and `raise` for exception raising:

```
def drunkFlip(amb: Amb, exc: Exc) = for {
  caught ← amb.flip()
  heads  ← if (caught) amb.flip() else exc.raise("Too drunk")
} yield if (heads) "Heads" else "Tails"
```

This example shows a few things

- The program is written in monadic style using Scala's `for`-comprehensions. Even though the program uses multiple effects, all effectful code only uses *one* monad – a variant of the continuation monad.
- The effect operations are methods on *capabilities* `amb` and `exc` which the method `drunkFlip` receives as arguments. The semantics of the effect operations is thus dependent on the corresponding implementations of `Amb` and `Exc`.

We can, for instance, run the method `drunkFlip` with the handlers `maybe` and `collect`:

```
val res1: List[Option[String]] = run {
  collect { amb ⇒ maybe { exc ⇒ drunkFlip(amb, exc) } }
}
//> List(Some(Heads), Some(Tails), None)
```

The `collect` handler enumerates all possible outcomes of the `flip` operation and collects them in a list. The `maybe` handler returns `None` if the program raises an exception.

Swapping the two handlers changes the result type and the semantics:

```
val res2: Option[List[String]] = run {
  maybe { exc ⇒ collect { amb ⇒ drunkFlip(amb, exc) } }
}
//> None
```

This illustrates an important feature of effect handlers. Programs that use effects are agnostic of the concrete handlers and their order (Plotkin & Pretnar, 2009). This gives the caller of the program and the implementer of the handlers more flexibility. Moreover, effect handlers are powerful enough to express many different control-flow structures as libraries, which otherwise have to be built into a language. Examples are async-await, cooperative multitasking, iterators, exceptions, and many more (Wu *et al.*, 2014; Leijen, 2016, 2017a; Dolan *et al.*, 2017).

In this paper, we present Effekt: a library for programming with effect handlers in the language Scala. The combination of effect handlers with object oriented features enables new modularization strategies, both for effectful programs and for effect handler implementations. A previous version of Effekt was briefly introduced (Brachthäuser & Schuster, 2017), but it lacked effect safety. The different aspects of our library design are summarized in the type signature of the method `drunkFlip` that we've seen above:

```
def drunkFlip(amb: Amb, exc: Exc): Control[String, amb.effect & exc.effect]
```

Result Type; Effect Typing; Capability passing style; Monad for Delimited Control

Our library design centers around the concept of *capability passing*. As we will see in the remainder of this paper, capabilities in Effekt encapsulate three different things:

Capabilities *contain* Effect Implementations They give semantics to effect operations (Section 2). In the example program `drunkFlip` we call effect operations as methods on the capabilities `amb` and `exc`, for instance.

Capabilities *contain* Prompt Markers Effect operations can capture the continuation delimited by the corresponding handler. Programs written with our library have type `Control` (Section 3), a monadic implementation of delimited control with first-class prompts (Dybvig *et al.*, 2007). Our capabilities contain such a prompt marker as a value member.

Capabilities *contain* Effect Labels Our capabilities contain a type member (`amb.effect`) that we use as a label to guarantee effect safety (Section 4). We keep track of all used capabilities by aggregating their effect members in an intersection type as the second type parameter of `Control`.

Our implementation of Effekt not only combines object oriented programming with effect handlers but also is the first practical implementation of an effect system with all of the following properties:

Effect Safety Our effect system asserts that all effects are handled. For example, we can only call `run` on a program when all effects are handled and we reject programs like `run { amb.flip() }`.

Effect Subtyping We use Scala's support for subtyping of intersection types to implement effect subtyping. A program with type `Control[Int, exc.effect]` can be used where a program of type `Control[Int, exc.effect & amb.effect]` is expected.

Effect Polymorphism We use Scala's support for polymorphism to express effect polymorphic functions like:

```
def map[A, B, E](ls: List[A], f: A ⇒ Control[B, E]): Control[List[B], E]
```

Here, `map` is polymorphic in the effects `E` used by function `f`.

Effect Parametricity Our effect system supports effect parametricity (Biernacki *et al.*, 2018). That is, looking at the type of `map` above we can guarantee that no implementation of `map` can (accidentally or purposefully) handle effects `E` used by `f`.

Effect Encapsulation Our effect system supports effect encapsulation (Lindley, 2018) – in variations also called *abstraction safety* (Zhang & Myers, 2019). Effectful higher order functions that use and handle effects locally don't leak these implementation details in their types.

Existing implementations of languages with effect handlers, either completely lack a static effect system – this includes MulticoreOCaml (Dolan *et al.*, 2014), Eff (Bauer & Pretnar, 2015), embeddings of Eff in OCaml (Kiselyov & Sivaramakrishnan, 2016), and previous versions of Effekt in Scala (Brachthäuser & Schuster, 2017) and Java (Brachthäuser *et al.*, 2018) – or they don't have sufficient support for effect polymorphism (Kammar *et al.*, 2013; Inostroza & van der Storm, 2018). Languages with effect systems like Extensible Effects (Kiselyov *et al.*, 2013), Koka (Leijen, 2014), Links (Hillerström *et al.*, 2017), and Frank (Lindley *et al.*, 2017) in turn don't support effect parametricity or require explicit lifting annotations to encapsulate effects. Effekt requires no such manual lifting.

In summary, our contributions are:

- For our implementation, we build on the operational semantics of Dybvig *et al.* (2007) but make it effect safe. To the best of our knowledge, we are the first to present an effect safe implementation of multi-prompt delimited control. We achieve effect safety by generalizing techniques of Launchbury & Sabry (1997) to nested regions (Kiselyov & Shan, 2008) but using intersection types of abstract type members (Parreaux *et al.*, 2018) instead of rank-2 types.
- We implement Effekt as a very thin layer on top of multi-prompt delimited continuations. Importantly, we demonstrate how effect safety for effects and handlers follows

from the newly gained effect safety for multi-prompt delimited continuations. The resulting effect language can be seen as a lightweight embedding into Scala of the language formally presented by Zhang & Myers (2019).
- We discuss interesting opportunities to explore type and effect safe modularization of effectful programs, opened up by embedding Effekt into Scala, a language that combines functional programming with object oriented programming.

The remainder of the paper is structured as follows. In Section 2 we give an overview of programming with Effekt and show how to implement effect handlers. While we ultimately aim to achieve an effect safe implementation of effect handlers, as an intermediate step we present a monadic implementation of delimited control (Section 3) to then show how to make it effect safe (Section 4). In Section 5, we express effect handlers as a very thin library on top of delimited control and discuss novel extensibility properties that arise from our embedding into Scala. Section 6 discusses related work and Section 7 concludes.

2. Programming with Effect Handlers in Effekt

To introduce programming with effect handlers in our library Effekt, we continue to use the effects from the introduction as a running example. We will now see how to declare and handle the exception and ambiguity effects. This section should give the reader a high-level intuition for the usage of the library. The `Control` monad is discussed in Section 3 and the effect system is described in Section 4. The examples of this section have been presented in similar form in Koka (Leijen, 2017b) and previous versions of Effekt for Scala (2017) and Java (2018). For syntactic convenience and better type inference, all code in this paper is given in Dotty, the upcoming next version of the Scala programming language. The library and the examples from this paper are available online:

`https://github.com/b-studios/scala-effekt/tree/jfp`

2.1. Exceptions

It might seem a bit contrived to implement exceptions using effect handlers in Scala since they are already built into the language. However, exceptions are the simplest algebraic effect but still provide a good overview over the involved concepts. There is also an important difference to exceptions built into Scala: our effect system guarantees that all effects are handled, which means that the exceptions we implement are checked. This is not the case for Scala's exceptions, they are unchecked.

Programming with effect handlers encourages modularity by separating the interface of an effect (the effect signature) from its implementation (the effect handler). Figure 1a declares the effect signature `Exc`, which inherits from the trait `Eff`:

```
trait Eff { type effect }
```

The library trait `Eff` provides us with the abstract type member `effect`. The signatures of effect operations like `raise` tell us not only their return type, in this case `Nothing`. They also taint the effect row, i.e. the right hand side in `Result / Effects`, by saying that they use the abstract type member `this.effect`. We use effect types like `this.effect` only for effect safety. They do not have any operational meaning attached to them and can be completely erased at runtime.

```
trait Eff { type effect }
trait Exc extends Eff { def raise(msg: String): Nothing / effect }
trait Amb extends Eff { def flip(): Boolean / effect }
```

(a) Effect signatures for exception and ambiguity as traits extending library trait `Eff`.

```
def maybe[R, E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E =
  // (1) delimit the scope of the handler
  handle {
    // (2) create handler instance / capability
    val exc = new Exc with Handler() {
      // (3) implement effect operations
      def raise(msg: String) = use { resume ⇒ pure(None) }
    }
    // (4) provide handled program with capability
    prog(exc) map { r ⇒ Some(r) }
    //       ^^^ (5) lift pure values into the effect domain
  }
```

(b) Handler function for the exception effect.

```
def collect[R, E](prog: (amb: Amb) ⇒ R / (amb.effect & E)): List[R] / E =
  handle {
    val amb = new Amb with Handler() {
      def flip() = use { resume ⇒ for {
        xs ← resume(true)
        ys ← resume(false)
      } yield xs ++ ys }
    }
    prog(amb) map { r ⇒ List(r) }
  }
```

(c) Handler function for the ambiguity effect.

Fig. 1. Using Effekt to declare and handle exception and ambiguity effects.

Let's consider the following program which uses a capability for the exception effect:

```
def div(x: Int, y: Int)(exc: Exc): Int / exc.effect =
  if (y == 0) exc.raise("y is zero") else pure(x / y)
```

For notational convenience, the result type of `div` makes use of the type alias:

```
type /[+Result, -Effects] = Control[Result, Effects]
```

In Scala, type constructors with two arguments can be used infix.

We can read the type signature of `div` as "provided with an exception capability `exc`, `div` computes an integer using the capability `exc`". While mentioning `exc` twice in the type signature might seem redundant, Effekt makes explicit what otherwise is conflated in existing effect languages.

- *Dynamic Effect Semantics*. Passing `exc` as parameter gives the program (term-level) access to the methods of `Exc`, in this case `raise`. Other languages with support for effect handlers perform an implicit lookup for a handler for an effect operation like

raise at runtime. In contrast, using Effekt, we explicitly pass effect handlers in the form of capabilities as parameters or store them in fields.

- *Static Effect Semantics*. When we use the effect operations of the exc capability, we have to mention its type member effect in the effect type. We will go into the details of the effect system in Section 4 – for now it is enough to understand that we guarantee effect safety by tracking all unhandled effects in the type parameter Effects of Control.

2.1.1. Handling Exceptions

The effect signature Exc only specifies the available effect operations. To give them a concrete interpretation we define a *handler function* with the following signature:

```
def maybe[R, E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E
```

Handler functions fulfill two purposes. Firstly, they provide capabilities (i.e. exc) to the handled program. The handled program prog can use the capability as well as other effects E to produce a result of type R. Secondly, handler functions remove the used effect (i.e. exc.effect) from the effect type. Hence, the effect type exc.effect & E of the handled program becomes E in the result type of maybe.

Operationally, the handler function interprets the effectful program which would compute a result of type R (mnemonic for "return type") into a new semantic domain of type Option[R], the *effect domain*. As seen earlier, programs that raise exceptions will be handled to return None. Programs that do not raise an exception return Some(result).

The handler is polymorphic in both the result type R of the handled program prog and in all other effects E that the program might use and which are not handled by maybe.

Remark We interchangeably use the terminology *capability* and *handler instance*. While "capability" puts a focus on the concept of entitling the holder to use an effect, "handler instance" highlights the fact that handlers are implementations of effect signatures.

Figure 1b uses our Effekt library to give the implementation of the handler function maybe. The handler instance exc extends both the effect signature Exc, as well as the library trait Handler. To implement the effect operations, handlers are able to utilize the instance method use provided by the library trait Handler. Specialized to this example, use has the type:

```
def use[A](body: (A ⇒ Option[R] / E) ⇒ Option[R] / E): A / exc.effect
//                ^^^^^^^^^^^^^^^^^^^^
//                  the continuation
```

Calling use in our implementation of raise captures the continuation and binds it to the identifier resume in the provided body. We discard the continuation and immediately return None. Discarding the continuation corresponds to the expected semantics of exceptions unwinding the stack.

The call to the library function handle delimits the scope of the continuation captured with use. That is, the continuation captured by the corresponding call to use contains all frames up to and including the call to handle. While at runtime prog might arbitrarily

install delimiters before calling `raise`, the connection between the two functions `use` and `handle` is *statically scoped*. We know that in all calls to `exc.raise` the continuation captured by `use` will be delimited by this lexically enclosing `handle`.

2.1.2. Return Clauses

Delimiting the scope with `handle` requires the result type of the passed program to match the effect domain of the handler – that is `Option[R]` in our case. For this reason, we are mapping over the result of the program to wrap it in `Some`. In other languages like Koka, Eff or Frank this lifting of the result type into the effect domain is typically performed by *return clauses*. In these languages, in addition to the implementation of effect operations every handler also has to implement the return clause (called `unit` in the following example), similar to:

```
object exc extends Exc with Handler() {
  def unit(r: R): Option[R] / E = pure(Some(r))
  def raise(msg: String) = use { resume ⇒ pure(None) }
}
```

This additional abstraction exists both for historical and technical reasons. Historically, algebraic effect handlers were conceived as a fold over the tree of computation operations (Plotkin & Pretnar, 2009). Return clauses are required to lift pure values into the domain of computations. It is certainly possible to express return clauses in terms of mapping over the result like:

```
handle { object h extends Handler() {...}; prog(h) flatMap { h.unit } }
```

However, until very recently, in languages like Koka one couldn't generally make this transformation since effect operations used in the return clause might accidentally be handled by the same handler that has the return clause (Leijen, 2018; Lindley, 2018). This is not an issue in Effekt, since the connection between handler and operations is established explicitly via capability passing, instead of performing a runtime lookup to the closest handler for a given effect signature. We think that it is an advantage of capability passing that return clauses are not required to be part of the user interface of handlers while maintaining the same expressiveness.

2.2. Ambiguity

Our interpretation of the exception effect discards the continuation of the program when it encounters a `raise` and immediately returns `None`. Not only do we have the option to discard the continuation, we can also call it multiple times as the next example will illustrate. Again, Figure 1a first declares the effect signature of the `Amb` effect. The implementation of the handler function `collect` in Figure 1c handles the ambiguity effect changing the result type of the program from `R` to `List[R]`:

```
def collect[R, E](prog: (amb: Amb) ⇒ R / (amb.effect & E)): List[R]
```

To implement the `flip` operation used in the introductory example and enumerate all possible results we call the continuation twice and concatenate the results of both calls. The type of the continuation here is `resume: Boolean ⇒ List[R] / E`, so calling it with

`true` and `false` gives us lists of type `List[R]`. We concatenate both lists. After providing the program `prog` with the capability `amb`, we lift the result type `R` to the effect domain `List[R]` by wrapping the result in a singleton list.

2.3. The Effekt Library

In this section, we have encountered the basic concepts of programming with effect handlers in Effekt. While all effectful computation happens in one monad (`Control`), programming with effect handlers encourages a modularization into the three components effect signatures, effectful programs, and effect handlers.

Effect signatures like `Exc` are interfaces containing methods marked as effectful with an abstract type member `effect`. This abstraction is very powerful – not only is the implementation of the method left abstract, but we also leave open which effects an implementation might use. In a concrete implementation, all effectful methods share the type member `effect` much like all methods of an object share the private state.

Effectful programs use effect operations by explicit capability passing. This has a number of advantages: in the presence of multiple instances of the same effect it is straightforward to pick a specific one; there is no runtime overhead for looking up the right handler implementation – calling an effect operation is just a dynamic dispatch; capability passing guarantees effect encapsulation (Lindley, 2018), that is, we avoid the problem of effects being accidentally handled. The downside of explicit capability passing is its verbosity. Like we did in previous versions of Effekt for Scala (Brachthäuser & Schuster, 2017), we could hide most of it using implicit parameters and implicit function types. However, in this paper we refrain from doing so to reduce cognitive overhead and focus on the aspect of effect safety.

Effect handlers provide semantics to effect operations. We need to distinguish three different aspects of an effect handler. The *handler function*, like `collect`, is a higher order function that provides an `amb` capability and removes the `amb.effect` from the effect type of the handled program. The *handler implementation* is a class implementing the effect signature. In our above example the `Amb` interface is implemented by an anonymous inner class `new Amb with Handler() { ... }`. The *handler instance*, like `amb`, is an instance of the handler implementation.

The remainder of this paper iterates the running example of this Section and introduces all mentioned types and library functions in three steps: Section 3 gives an overview over the underlying implementation of delimited control and shows how programming with our interface for delimited control is already very close to programming with (algebraic) effects. Section 4 then establishes effect safety for this implementation of delimited control and introduces the abstraction of effect signatures. Finally, Section 5 introduces the rest of the functions and types (e.g. `handle` and `Handler`) that make it possible to program in the style of effects and handlers.

3. Delimited Control

Effekt implements effect handlers in terms of a monad for delimited control. In this section, we present a simplified version of our implementation of this monad as a specialization of the one presented by Dybvig *et al.* (2007).

> *"Effect handlers are to delimited continuations as structured programming is to goto"* – Andrej Bauer (Dagstuhl Seminar, March 2018).

For multiple decades control operators like `call/cc` have been used to program with control effects. Similar to `goto`, which can be understood as undelimited, local continuation (Landin, 1965; Kennedy, 2007), `call/cc` captures the (global) undelimited continuation. Again like `goto`, while being very expressive, programs written with the control operator `call/cc` tend to be fragile, hard to understand and maintain.

Recently, in disguise, control operators have found their way into mainstream programming languages as async/await, generators and other specialized solutions. At the same time, the programming languages research community found new interest in control effects in the form of algebraic effects and handlers. Delimited control operators and effect handlers are closely related. In the literature, effect handlers are often introduced as a structured way to program with delimited continuations (Kammar *et al.*, 2013; Leijen, 2017b). It also has been established practically (Kiselyov & Sivaramakrishnan, 2016) as well as theoretically (Forster *et al.*, 2017) that certain forms of delimited continuations can express certain forms of algebraic effect handlers. It is natural to base an implementation of effects and handlers on delimited continuations, reusing existing work on the latter.

We believe the regained interest comes from four important generalizations and improvements over `call/cc`:

1. generalizing from undelimited to delimited continuations
2. generalizing from one control operator to a family of control operators
3. establishing answer type safety of control operators
4. establishing effect safety of control operators

From an engineer's perspective, each of these improvements helps to write programs in a modular way making them easier to extend and making it easier to reason about parts of a program in isolation.

The version of our monad for delimited control that we introduce in this section is answer type safe, but it is not effect safe. In the next section, we show how to establish effect safety.

3.1. Delimiting Continuations

The following example program uses the control operator shift_0 (Kammar *et al.*, 2013) and delimiter reset:

$$1 + \mathsf{reset}\,\{\, 10 + \mathsf{shift}_0\,\{\, \lambda k.\, k(k(100))\,\}\,\}$$

The control operator shift_0 captures the continuation and binds it to `k`. The continuation is delimited: only the evaluation context up to the enclosing reset is captured and thus the

continuation corresponds to $10 + \square$. This example reduces in the following steps:

$$
\begin{array}{lll}
1 + \mathsf{reset}\,\{\, 10 + \mathsf{shift}_0\,\{\, \lambda k.\, k(k(100))\,\}\,\} & & \\
1 + & k(k(100)) & \textbf{where} \quad k = \lambda x.\, \mathsf{reset}\,\{\, 10 + x\,\} \\
1 + & k(\mathsf{reset}\,\{\, 10 + 100\,\}) & \\
1 + & k(110) & \\
1 + & \mathsf{reset}\,\{\, 10 + 110\,\} & \\
1 + & 120 & \\
121 & &
\end{array}
$$

The continuation k does *not* contain the frame $1 + \square$, which is outside of the delimiting reset. The body of the continuation k is again delimited by reset.

3.1.1. The `Control`-monad

In previous sections, we used a monad `Control[+Result, -Effects]` that is both answer-type safe and effect safe. To focus on the operational semantics of delimited control, in this section we start with a simpler variant `Control[+Result]` that has the same operational semantics, is answer-type safe but not effect safe. One can view `Control` as an embedding of a language with control effects into Scala. Our monad `Control[+Result]` is very close to the monad for multi-prompt delimited control by Dybvig *et al.* (2007), but we specialize the exposed interface to better fit effect handlers. While Dybvig *et al.*, present a very general framework that allows for the implementation of all sorts of control operators, we build on shift_0 as our control operator of choice which means that the body of the captured continuation is always delimited by a reset. As highlighted by Kammar *et al.* (2013), shift_0 matches the semantics of *deep handlers* where the same effect is already handled in the continuation. As we will see later, `shift`$_0$ has a very natural type in the presence of multiple delimiters (Schuster & Brachthäuser, 2018). It additionally avoids the problem of accumulating delimiters observed by Dybvig *et al.* (2007).

Example 1

The above direct style program translates to Scala using our `Control` monad as follows:

```
val ex: Control[Int] = reset { p ⇒
  shift0(p) { k ⇒ k(100) flatMap k } map { 10 + _ }
} map { 1 + _ }
```

Our delimiter `reset` introduces a *fresh prompt* `p`. Our control operator `shift`$_0$ takes a prompt as a parameter that allows us to select which `reset` we want to shift to. The current example only uses one prompt `p` but we will shortly see how to utilize this additional expressivity.

Figure 2 defines the interface of our monad for delimited control. As usual, we embed a pure value into the monad with `pure` and we sequence effectful computations with `flatMap`. This enables us to write effectful programs in an imperative style via Scala's for-comprehensions. In addition to being a monad, the type `Control` provides us with three operations. Using `run` we execute a program with control effects that computes a value of type `A` to obtain that value. The operation `reset { p ⇒ PROG }` delimits the control

```
trait Control[+A] {
  def flatMap[B](f: A ⇒ Control[B]): Control[B]
  def map[B](f: A ⇒ B): Control[B]
}

def pure[A](value: A): Control[A]
def run[A](program: Control[A]): A

trait Prompt[Result] { }
def reset[R](program: Prompt[R] ⇒ Control[R]): Control[R]

type CPS[A, R] = (A ⇒ R) ⇒ R
def shift0[A, R](prompt: Prompt[R])(body: CPS[A, Control[R]]): Control[A]
```

Fig. 2. The control monad without effect typing.

effects in the provided program `PROG`. It introduces a fresh prompt marker `p` and provides it to its argument. The control operator `shift0(p) { k ⇒ PROG }` receives a prompt marker `p` and a body `k ⇒ PROG`. It captures the current continuation up to the corresponding `reset` and passes the continuation to the body.

3.2. Multiple Prompts and Families of Control Operators

Every call to `reset` introduces a fresh prompt `p`, or in other words, each prompt `p` labels the corresponding `reset`. This gives rise to a dynamic number of control operators `shift(p)`, one for each prompt created by a `reset`. The following example illustrates the use of multiple resets:

Example 2

We use `reset` twice, introducing two different prompts p_1 and p_2.

```
val ex2: Control[Int] = reset { p1 ⇒
  reset { p2 ⇒
    shift0(p1) { k ⇒ pure(21) }
  } map { if (_) 1 else 2 }
} map { 2 * _ }
```

The captured continuation `k` contains the program segment delimited by prompt p_1. It corresponds to the evaluation context if $(\text{reset}\,\{\,\lambda p_2.\,\square\,\})\,1\,\text{else}\,2$. In this example the body of $\texttt{shift}_0$ discards the continuation `k` and immediately returns `21`. Hence, `run { ex2 }` evaluates to `42`.

3.3. Answer Type Safety

Operationally, `shift0(p)(k ⇒ PROG)` replaces the corresponding reset by the body `PROG`. The return type of the body has to be the same as the answer type at the `reset`. In our setting

of multiple first-class prompts we guarantee this by following Dybvig *et al.* (2007) and parametrizing prompts over the answer type `R` (Figure 2). In Example 2, the two prompts thus have types $\texttt{p}_1$`: Prompt[Int]` and $\texttt{p}_2$`: Prompt[Boolean]`.

In the type of `reset` we make sure that three types coincide to be type `R`: the answer type of the created prompt, the result of the given program, and the return type of `reset`.

In the type of $\texttt{shift}_0$ we then use the answer type of the given prompt to make sure that the return type of the continuation and the return type of the given body agree with the answer type expected at the `reset` that originally created the prompt `p`.

The continuation in Example 2 thus has type `k : Boolean ⇒ Control[Int]` and the body of the shift is expected to return `Control[Int]`. Answer type safety is especially important in the presence of multiple prompts. Each reset might introduce a prompt with a different answer type. Shifting to a reset with the wrong type should be ill-typed. For instance shifting to $\texttt{p}_2$ would render the example type incorrect since this would require the body of shift to return a computation of type `Boolean`, not `Int`.

3.4. Programming with Delimited Control

To highlight how programming with effect handlers is structured programming with delimited control, we express our running example directly in terms of multi-prompt delimited control.

Example 3

Figure 3 translates our running example from Figure 1 to directly use control effects. Let us assume the type aliases for effectful functions with the signatures of `flip` and `raise` (Figure 3a). The user program from the introduction then carries over almost unchanged.

```
def drunkFlip(raise: Exc, flip: Amb): Control[String] = for {
  caught ← flip()
  heads  ← if (caught) flip() else raise("Too drunk")
} yield if (heads) "Heads" else "Tails"
```

The function `drunkFlip` now takes effectful functions, or if you will *effect operations*, `raise` and `flip` directly as parameters.

We implement what could be viewed as handlers for `raise` (Figure 3b) and `flip` (Figure 3c) as higher order functions that construct implementations of effect operations and pass them to the given program. They also change the result type `R` to the effect domain `Option[R]` and `List[R]` respectively, just as we have previously seen. Following this pattern to directly implement effect handlers in terms of delimited control shows that handler functions encapsulate three aspects of effect handling in one module:

1. the handler function uses `reset` to delimit the scope of the captured continuation;
2. it locally uses the fresh prompt introduced by `reset` to implement the effect operations in terms of $\texttt{shift}_0$ – the effect operations close over the prompt and are thus the only way to capture the continuation;
3. it finally lifts the return type of the handled function `R` into the effect domain which makes it the answer type of the `reset`.

```
type Amb = () ⇒ Control[Boolean]
type Exc = String ⇒ Control[Nothing]
```

(a) Effect signatures for exception and ambiguity as type aliases for effectful functions.

```
def maybe[R](prog: Exc ⇒ Control[R]): Control[Option[R]] = reset { p ⇒
  val raise: Exc = msg ⇒ shift0(p) { resume ⇒ pure(None) }
  prog(raise) map { x ⇒ Some(x) }
}
```

(b) Handler function for the exception effect.

```
def collect[R](prog: Amb ⇒ Control[R]): Control[List[R]] = reset { p ⇒
  val flip: Amb  = () ⇒ shift0(p) { resume ⇒ for {
    xs ← resume(true)
    ys ← resume(false)
  } yield xs ++ ys }

  prog(flip) map { x ⇒ List(x) }
}
```

(c) Handler function for the ambiguity effect.

Fig. 3. Using answer type safe delimited control to declare and handle exception and ambiguity effects.

Grouping these aspects of effect handling in one module, it is possible to locally reason about type safety. The implementation of `raise` is only safe because we statically know from the type of `p: Prompt[Option[R]]` that the answer type expected at the `reset` is `Option[R]`. Likewise, in `collect` we use the fact that we statically know that the answer type in the body of $\texttt{shift}_0\texttt{(p)}$ is `List[R]` to safely concatenate the results of the two calls of the continuation `resume`.

Just as in the introduction, we can use both handler functions in different order to run the program, getting different results of different type.

```
val res1: List[Option[String]] = run {
  collect { flip ⇒ maybe { raise ⇒ drunkFlip(raise, flip) }}
}
val res2: Option[List[String]] = run {
  maybe { raise ⇒ collect { flip ⇒ drunkFlip(raise, flip) }}
}
```

The operations `raise` and `flip` are first-class functions and close over the fresh prompts `p` that we introduced with `reset`. However, if they escape the scope of the corresponding handler function, calling them will lead to a runtime exception. The next section addresses this source of unsafety among others.

4. The Effect System

In the previous section, we have seen a version of `Control` that has a type parameter `Result`. By also indexing `Prompt` with a type parameter `Result`, we statically track answer types and guarantee that capturing and calling the continuation is type safe. However, as we will see shortly, this version of `Control` is not effect safe and using a prompt outside of the dynamic scope of the corresponding `reset` leads to a runtime error. We identify two ways to leave the scope of `reset`.

Leaving the scope by returning We leave the scope of `reset` by returning from it. In this case it is possible to leak the prompt introduced by `reset` either through the heap

```
var o: Prompt[Unit] = null
val problem1 = run {
  for {
    _ ← reset { prompt ⇒ p = prompt; pure(()) }
    _ ← shift0(p) { resume ⇒ pure(()) } // Exception: Prompt not found
  } yield ()
}
```

or by simply returning it:

```
val problem2 = run {
  for {
    p ← reset { p ⇒ pure(p) }
    _ ← shift0(p) { resume ⇒ pure(()) } // Exception: Prompt not found
  } yield ()
}
```

Both sources of leakage can also occur indirectly through values that close over the prompt, like the effect operations in the previous section. Prompts might even leave the scope of the enclosing `run` to then be used in the scope of a different `run`. Dybvig *et al.* (2007) use rank-2 types to prevent this particular source of error, but leave others to future work.

Leaving the scope by shifting We can also leave the scope of `reset` by means of control effects.

```
val problem3 = run {
  reset { p ⇒
    shift0(p) { resume ⇒
      shift0(p) { resume ⇒ pure(()) } // Exception: Prompt not found
    }
  }
}
```

Since shifting to a prompt removes the enclosing `reset`, shifting a second time inside of the body of $\texttt{shift}_0$ will result in a runtime error. The captured continuation is not delimited anymore. Danvy & Filinski (1990) operationally prevent this kind of runtime error by leaving the `reset` behind. However, the delimiter can also be removed from the stack by continuation capture:

```
type Pure = Any
type /[+A, -E] = Control[A, E]

trait Control[+A, -Effects] {
  def flatMap[B, E](f: A ⇒ B / E): B / (E & Effects)
  def map[B](f: A ⇒ B): B / Effects
}

def pure[A](value: A): A / Pure
def run[A](c: A / Pure): A

trait Prompt[Result, Effects]
def reset[R, E](prog: (p: Prompt[R, E]) ⇒ R / (p.type & E)): R / E

type CPS[A, R] = (A ⇒ R) ⇒ R
def shift0[A, R, E](p: Prompt[R, E])(body: CPS[A, R / E]): A / p.type
```

Fig. 4. The control monad with effect typing.

```
val problem4 = run {
  reset { p1 ⇒ reset { p2 ⇒
    shift0(p1) { resume ⇒
      shift0(p2) { resume ⇒ pure(()) } // Exception: Prompt not found
    }
  }}
}
```

We now introduce an effect system that rules out the above four problem programs, prevents the use of escaped prompts and guarantees effect safety. To the best of our knowledge, Effekt is the first type *and effect safe* embedding of multi-prompt delimited control. The underlying problem our effect system solves is a very general one: we need to restrict the lifetime of a resource (prompts in our case) to a certain dynamic region (`reset` in our case). This problem occurs in the domain of region based resource management (Kiselyov & Shan, 2008), object capabilities (Haller & Loiko, 2016), delimited control (Dybvig *et al.*, 2007), macro hygiene (Parreaux *et al.*, 2018) as well as with prompt based implementations of effect handlers (Brachthäuser & Schuster, 2017).

4.1. Tracking and Delimiting Prompt Usage

Our effect system builds on the idea of tracking the set of prompts used by a program in the type of the program. To enable tracking of effects, Figure 4 thus defines our final version of `Control` with a second type parameter `Effects`. We represent prompts on the type-level by their singleton types and we use Scala's intersection types to describe a set of prompts. Crucially, we also generalize the answer type – it is now effectful: prompts track both, the expected return type and the set of prompts in scope at the corresponding `reset`.

As an example, assuming prompts p_1, p_2, and p_3 we write the type of the effectful program that uses prompts p_1 and p_2 to compute an integer as

```
val prog1: Control[Int, p1.type & p2.type]
```

Here, `p1.type` is the singleton type of the prompt `p1` and `p1.type & p2.type` is an intersection type. In general, the intersection of singleton types might not be inhabited, but this is irrelevant for our use case, since we only use the intersection type of prompts as a phantom type to track used effects.

To support effect subtyping, the type parameter `Effects` of `Control` is marked as contravariant.

```
val prog2: Control[Int, p1.type & p2.type & p3.type] = prog1
```

The above assignment is type correct since we have by subtyping:

```
p1.type & p2.type & p3.type <: p1.type & p2.type
```

We define `type Pure = Any`, where `Any` is the top of the Scala subtyping lattice. Pure programs have an effect type `Pure` since they do not use any prompts. By contravariance they are a subtype of effectful programs that use a non-empty intersection of prompt types.

We do not prevent leakage of prompts. Instead, we taint the effect type whenever we perform $\texttt{shift}_0$ on a prompt. This is reflected in the return type of $\texttt{shift}_0$ (Figure 4). The type `A / p.type` indicates the use of the prompt `p` on the type level.

While we do not prevent leakage, the type of `run` asserts that only pure programs can be executed. That is, all prompts have to be delimited and the intersection has to be empty (`Pure`). Conceptually, this is similar to how rank-2 types can be used to enable type safe monadic regions in Haskell (Launchbury & Sabry, 1997; Kiselyov & Shan, 2008). However, since rank-2 types are not well-supported in Scala, we use singleton types for ergonomics and better type inference.

To guarantee safety, we have to make sure that the only way to remove a prompt type from the intersection is by delimiting the program with `reset`. Our `reset` has the following signature:

```
def reset[R, E](prog: (p: Prompt[R, E]) ⇒ R / (p.type & E)): R / E
```

Here, `prog` has a dependent function type: the return type is (path) dependent on its value parameter `p`. Different calls to `reset` lead to different singleton types. Hence, only the `reset` that introduced a prompt can remove its very own singleton type from the effect type. Again, this is close to rank-2 types, but moves the universal quantification from the type level to the term level. This excludes problematic programs like $\texttt{problem}_1$ and $\texttt{problem}_2$ from above.

Remark In Scala, two (path-dependent) singleton types are equal if and only if their prefix paths are stable and they can be unified (Odersky & Zenger, 2005b). Informally, a path is stable if it does not contain a mutable component. This way we prevent leakage via mutable references as in $\texttt{problem}_1$.

4.2. From Answer Type Safety to Effect Safety

In the previous variant of `Control`, prompts carried the answer type `Result` to ensure that using control effects is type safe. To also make them effect safe and prevent programs like

$\texttt{problem}_4$ from type checking, the type `Prompt` (Figure 4) now additionally contains a type parameter `Effects`.

```
trait Prompt[Result, Effects]
```

The type `Prompt[R, E]` can conceptually be understood as type `Prompt[R / E]`. However, we track the two aspects in separate type parameters to improve type inference.

Intuitively, the body of a $\texttt{shift}_0$ is evaluated at the position of the corresponding `reset`. This is reflected in its type which expands to:

```
body: (A ⇒ R / E) ⇒ R / E
```

Both, the answer type `R` and the effects `E` have to match with the ones at the corresponding `reset[R, E]`. Thus prompt-passing style is not only essential for operationally delimiting control effects but also necessary to carry both the expected answer type as well as the available effects from the `reset` to the $\texttt{shift}_0$ that uses the prompt.

Since the body of $\texttt{shift}_0$ has to return `R / E` it cannot shift to the same prompt (as in $\texttt{problem}_3$). This would require a type of `R / (p.type & E)`. The problematic program $\texttt{problem}_4$ is ruled out too, since $\texttt{p}_1$ has type `Prompt[Int, Pure]` and thus the body of the first shift needs to be pure and cannot use $\texttt{p}_2$.

Example 1 - Effect Typed

We are now ready to revisit the examples from the previous section and assign effect types. The first example does not need to change. Only the type is a bit more precise:

```
val ex: Int / Pure = reset { p: Prompt[Int, Pure] ⇒
  shift0(p) { k ⇒ k(100) flatMap k } map { 10 + _ }
} map { 1 + _ }
```

It is now clear from the effect type that after resetting, there are no more control effects left to delimit and we can safely run `ex`.

Example 2 - Effect Typed

The second example illustrates how each `reset` removes its corresponding prompt from the effect type.

```
val ex2: Int / Pure = reset { p1: Prompt[Int, Pure] ⇒
  reset { p2: Prompt[Boolean, p1.type] ⇒
    shift0(p1) { k ⇒ pure(21) }  // Control[Int, p2.type & p1.type]
  } map { if (_) 1 else 2 }      // Control[Int, p1.type]
} map { 2 * _ }                  // Control[Int, Pure]
```

While the program only shifts to prompt $\texttt{p}_1$, the type of `reset` requires the body of the inner `reset` to have type `Control[Int, p1.type & p2.type]` which it has by effect subtyping.

Example 3 - Effect Typed

Adding effect types to the third example is a bit more involved. In the previous section, we defined type aliases as interfaces of the effectful functions `flip` and `raise` (Figure 3a).

```
trait Eff { type effect }
trait Exc extends Eff { def raise(msg: String): Nothing / effect }
trait Amb extends Eff { def flip(): Boolean / effect }
```

(a) Effect Signatures for exception and ambiguity.

```
def maybe[R, E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E =
  reset { p ⇒
    val exc = new Exc {
      type effect = p.type // type refinement
      def raise(msg: String) = shift0(p) { ... }
    }
    prog(exc) map { x ⇒ Some(x) }
  }
```

(b) Handler function for the exception effect. Implementation of `raise` like in Figure 3b.

```
def collect[R, E](prog: (amb: Amb) ⇒ R / (amb.effect & E)): List[R] / E =
  reset { p ⇒
    val amb = new Amb {
      type effect = p.type // type refinement
      def flip() = shift0(p) { ... }
    }
    prog(amb) map { x ⇒ List(x) }
  }
```

(c) Handler function for the ambiguity effect. Implementation of `flip` like in Figure 3c.

Fig. 5. Using effect safe delimited control to declare and handle exception and ambiguity effects.

While then it was sufficient to say that `flip` and `raise` use *any* control effects by making them return for example `Control[Boolean]`, we now have to be more specific. The traits `Amb` and `Exc` in Figure 5a are not much different from the equally named type aliases we defined earlier. The two differences are:

1. Operations are now named (that is `flip` and `raise` are explicitly named methods) whereas earlier we used the `apply` method that Scala generates for function types.
2. Guided by the implementation of the previous section, we know that we will eventually use *some prompt* to implement each of the effect operations. Since we don't want to expose implementation details, we hide the prompt type behind an existentially qualified type member `effect`.

Maybe not surprisingly, these types coincide with the definitions of effect signatures in Figure 1a. The use of these effects is now exactly as in the introductory example as well:

```
def drunkFlip(exc: Exc, amb: Amb): String / (amb.effect & exc.effect) =
  for {
    caught ← amb.flip()
    heads  ← if (caught) amb.flip() else exc.raise("Too drunk")
  } yield if (heads) "Heads" else "Tails"
```

The implementation of handler functions is given in Figures 5b and 5c. The implementations of `raise` and `flip` in the handler functions are like in the previous section and we omit them. However, we do have to assign more precise types! Most importantly, handlers now have to establish the type equivalence between `type effect` and the singleton type of the prompt `p.type` that they use in the implementation of the effectful methods `raise` and `flip`. The type refinement `type effect = p.type` is necessary to unify `amb.effect` with `p.type`. This way `reset` removes `amb.effect` from the effects.

With types assigned to `maybe` and `collect`, we are ready to type and effect check the example program from the introduction:

```
val res1 = run {
  collect { amb ⇒
    maybe { exc ⇒
      drunkFlip(amb, exc)    // Control[String, exc.effect & amb.effect]
    }                        // Control[Option[String], amb.effect]
  }                          // Control[List[Option[String]], Pure]
}                            // List[Option[String]]
```

4.3. Effect Parametricity

Since we embed our effect system into Scala, we can reuse Scala's support for subtyping and type polymorphism to express effect subtyping and effect polymorphic functions. This is an important advantage over effect systems that encode effect rows using type level lists. One example of an effect polymorphic, higher-order function is

```
def map[A, B, E](lst: List[A], f: A ⇒ B / E): List[B] / E
```

The function `map` is effect polymorphic in the effects `E` used by function `f`. The return type of `map` indicates that it potentially calls `f` in its implementation and so has the same effects `E` as `f`. The effects still need to be handled by the caller of `map`. In particular, since `map` is polymorphic in the effects `E`, we claim that it *should not be possible* for it to (accidentally) handle any concrete effect in `E` no matter what `E` will be instantiated to at the call site. That is, in the following user program, we should be able to determine *statically* that `flip` is handled by `collect` and no implementation of `map` should be able to violate this assumption.

```
collect { amb ⇒ map(List(1,2,3), n ⇒ amb.flip()) }
```

We refer to this property as *effect parametricity*. It has also been called *abstraction safety* in the literature (Zhang & Myers, 2019).

4.3.1. Effect Encapsulation

A variant of this problem can be observed in languages featuring an ML-like type system with row-polymorphism for effect types like Koka and Frank – referred to as the *effect encapsulation* problem (Lindley, 2018; Leijen, 2018).

The problem is illustrated in the following program adapted from Leijen (2018) written in the Koka language.

```
fun f(action: () → <exc|e> a): e option<a> { // types inferred
  maybe {
    if (...) { raise("abort") }
    action()
  }
}
```

Here, `f` is a higher-order function that takes an effectful function `action` as its argument. The function `f` uses exceptions in its implementation but locally handles them with the `maybe` handler. This implementation detail still *leaks* as part of the inferred type. The inferred type now states that `exc` effects of `action` will be handled by `f`. The reason is that Koka implements effect subtyping via row polymorphism so the effect row of `action()` needs to be unified with the other statements handled by `maybe`.

4.3.2. Accidental Handling

We might try to annotate `action` with the type `() → e a` but this will not type check. And rightfully so: types and effects in Koka are erased at compile time and do not influence the runtime semantics. Removing `exc` from the row of effects of `f` hides the fact that the operational semantics of Koka will handle any exception effect used in `action` with the `maybe` handler in `f`.

As a solution to this problem Koka and Frank introduce some form of manual lifting operation (Biernacki *et al.*, 2018). Using `inject`, in Koka the above example can be rewritten to

```
fun f(action: () → e a): e option<a> { // types inferred
  maybe {
    if (...) { raise("abort") }
    inject<exc> { action() }
  }
}
```

Manually injecting the `exc` effect into the effect row also has operational content as described by Leijen (2018): the runtime search for the exception handler will skip this `maybe` handler in `f`.

Since they are based on runtime lookup of a handler for a given global effect, the accidental handling of effects can occur in languages without static effect systems like MulticoreOCaml (Dolan *et al.*, 2014) and Eff (Bauer & Pretnar, 2015). The previous presentation of Effekt for Scala (Brachthäuser & Schuster, 2017) never had this encapsulation problem and was prepared to support effect parametricity. Just like presented in this paper

- user programs are written in capability passing style – that is capabilities are explicitly referenced and not looked up at runtime;
- every reset introduces a fresh prompt at runtime – this avoids accidental capture even if the same handler function is called multiple times.

The effect system presented in this section is designed to integrate well with the existing operational semantics. The combination of capture free operational semantics and effect safety gives us effect parametricity. A theoretical result about effect parametricity has been

presented recently by Zhang & Myers (2019). We present a practical implementation of a very similar effect system in Scala.
Concretely, in Effekt we can write two variants of the function `f` with different types:

```
def f1[A, E](action: (exc: Exc) ⇒ A / (exc.effect & E)): Option[A] / E
def f2[A, E](action: () ⇒ A / E): Option[A] / E
```

The type of f_1 makes clear that `action` has an unhandled exception effect that f_1 might handle. In the second variant, f_2 is fully parametric in the effects used by `action`. No effect handler in f_2 can interfere with the effects used by `action`.

5. Even More Extensible Effects

In the previous section we have seen how to add effect safety to a library for programming with delimited control and multiple, first-class prompts in Scala. When comparing the implementations of handler functions `maybe` and `collect`, we find that there is only a small difference between programming with multi-prompt delimited control (Section 4) and programming with effect handlers (Section 2). In this section, we introduce the missing interfaces to program with effect handlers, highlight extensibility properties of our implementation, and discuss the combination of effect handlers and object oriented programming.

5.1. From Delimited Control to Effect Handlers

We purposefully presented programming with delimited control close to programming with effect handlers to highlight one small, but important difference:

Effect handlers encapsulate the introduction of a prompt and its use.

What we mean is that users don't manually pass prompts around which then can be used to capture the continuation at arbitrary points in the program. Instead, there is a lexical relation between effect operations that use a prompt and the `reset` that introduced the prompt. We believe that this is one of the most important aspects that make programming with effect handlers more approachable than programming with (multi-prompt) delimited control.

The following example repeats the implementation of handler function `maybe` from Section 2 but makes it explicit that handlers close over prompts.

```
def maybe[R, E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E =
  handle { implicit prompt ⇒
    val exc = new Exc with Handler(prompt) {
      def raise(msg: String) = use { resume ⇒ pure(None) }
    }
    prog(exc) map { r ⇒ Some(r) }
  }
```

The `handle` function and the trait `Handler` are both defined in Figure 6. Like `reset`, the `handle` function brings a fresh prompt into scope. The trait `Handler` takes a prompt as its

```
def handle[R, E](prog: implicit (p: Prompt[R, E]) ⇒ R / (p.type & E)): R / E
  = reset(p ⇒ prog(p))

trait Use[R, E] extends Eff {
  def use[A](body: CPS[A, R / E]): A / effect
}

trait Handler[R, E](implicit val prompt: Prompt[R, E]) extends Use[R, E] {
  type effect = prompt.type
  def use[A](body: CPS[A, R / E]) = shift0(prompt)(body)
}
```

Fig. 6. Effect handlers in terms of delimited control.

constructor argument and closes over it by storing it in the field `prompt`. It then implements `use` in terms of `shift0(prompt)`. Our effect handler interface hides the implementation in terms of multi-prompt delimited control by making the prompt-passing implicit.

We split capturing the continuation with `use` into an interface `Use` and an implementation `Handler`. This way, we can factor the `maybe` and `collect` handler implementations into separate, reusable traits:

```
trait Maybe[R, E] extends Exc with Use[Option[R], E] {
  def raise(msg: String) = use { resume ⇒ pure(None) }
}
trait Collect[R, E] extends Amb with Use[List[R], E] {
  def flip() = use { resume ⇒ for {
    xs ← resume(true)
    ys ← resume(false)
  } yield xs ++ ys }
}
```

Handling the `Exc` effect with the `Maybe` handler trait now amounts to constructing a handler instance of `Maybe` and passing it to the program.

```
handle { prog(new Maybe[R, E] with Handler()) map { r ⇒ Some(r) } }
```

The handler instance implicitly closes (via `Handler()`) over the prompt which in turn is implicitly brought into scope by `handle`.

Defining handlers as traits allows us to use mixin composition and thereby discover new opportunities for extensible handler definitions which we explore in the remainder of this section.

5.2. The Effect Expression Problem

Most implementations of libraries and languages for (algebraic) effects and handlers are based on a *deep embedding* of effect operations. They reify effect operations as alternatives in a sum type and represent effectful computations as a command-response tree. For

instance, the `flip` effect operation would be reified as a constructor of an algebraic data type `Amb`. Handlers fold over the tree of computation and use pattern matching to interpret the reified effect operations (Leijen, 2014; Hillerström *et al.*, 2017; Bauer & Pretnar, 2015; Kiselyov & Ishii, 2015; Kiselyov & Sivaramakrishnan, 2016). To mix programs with different effects means to extend an open union type of reified effect operations.

In contrast, by performing capability passing, Effekt builds on a *shallow embedding* (Hudak, 1998; Carette *et al.*, 2007) of effect operations. Instead of folding over the tree of computation, user programs directly call effect operations on the handler. In a language with mixin composition, shallow embeddings can be structured in a pleasingly extensible way (Oliveira & Cook, 2012). Thus, Effekt has a solution to the expression problem (Wadler, 1998) at its foundation, a property it shares with many other effect handler implementations. For instance languages like Koka (Leijen, 2014), Frank (Lindley *et al.*, 2017), and Links (Hillerström *et al.*, 2017) are based on row polymorphism (Gaster & Jones, 1996) and Extensible Effects (Kiselyov *et al.*, 2013; Kiselyov & Ishii, 2015) are based on open unions (Swierstra, 2008).

Viewing the tree of computation as a recursive data type, we can describe the *effect expression problem* (Brachthäuser & Schuster, 2017) as modularly and typesafe being able

a. to implement new handlers for an effect operation – this corresponds to adding a new function definition over the recursive data type in the original expression problem;
b. to add new effect operations – this corresponds to adding a new variant to the recursive data type in the original expression problem.

The analogy to the expression problem, however, is not perfect: Most descriptions of the expression problem only consider a single algebra, whereas with effect handlers we typically have more than one effect signature and the order of handling / folding over the operations affects the semantics.

5.2.1. Dimensions of Extensibility

We can relate extensibility dimensions discussed in the literature on the expression problem to the effect handler setting and show how Effekt supports them. Importantly, by embedding effect handlers into a general purpose programming language like Scala, the modularity features of the host language become available to structure effectful programs and handlers.

Adding new handlers for an effect. The first dimension of the effect expression problem. A central feature of every implementation of effects and handlers is the ability to define a new handler for an existing effect. We support this feature: users can define a new trait that implements an existing effect signature.

Adding new operations to an effect. The second dimension of the effect expression problem. It is important to distinguish adding an operation to an existing effect signature and adding a new effect signature. Effekt supports both in a modular way as required for solutions to the expression problem. While Section 2 already illustrated how to define new effect signatures, let us look at an example of how to extend the effect signature of `Amb` with a nondeterministic choice operator:

```
trait Choose extends Eff {
  def choose[A](first: A, second: A): A / effect
}
trait AmbChoose extends Amb with Choose
```

The definition of the new effect signature `AmbChoose` introduces a subtyping relationship between `Amb` and `AmbChoose` – in consequence, programs that use the `Amb` effect can also be handled by a handler supporting `AmbChoose`. To implement the handler, we could for example mix in `Collect` and implement the new effect operation `choose` in terms of effect operation `flip` that we have already implemented in `Collect`.

```
trait CollectChoose[R, E] extends AmbChoose with Collect[R, E] {
  def choose[A](first: A, second: A): A / effect = for {
    b ← flip()
  } yield if (b) first else second
}
```

Combining independently developed effect signatures and handlers. The description of the expression problem has seen many extensions and additional requirements. One additional requirement described by Odersky & Zenger (2005a) is that the programmer should be able to combine independently developed extensions. This requirement might seem unnecessary in the context of effect handlers since instead of combining two effect signatures a user can just use both effects separately. However, using trait mixin composition to combine two handlers, the handler implementations can share the effect domain as well as implementation details like private methods and dependencies on other internally used effects.

```
trait Backtrack[R, E] extends Amb with Use[Option[R], E] {
  def flip() = use { resume ⇒ for {
    fst ← resume(true)
    res ← if (fst.isDefined) pure(fst) else resume(false)
  } yield res }
}
trait Both[R, E] extends Backtrack[R, E] with Maybe[R, E]
def both[R, E](prog: (b: Amb & Exc) ⇒ R / (b.effect & E)): Option[R] / E =
  handle {
    prog(new Both[R, E] with Handler()) map { r ⇒ Some(r) }
  }
```

The handler `Backtrack` is an alternative handler for `Amb`, interpreting ambiguity into the effect domain `Option[R]`. Since the effect domains coincide, `Both` can mix `Backtrack` and `Maybe` in one handler. Using the handler function `both`, we can handle `Exc` and `Amb` simultaneously as in:

```
val res3: Option[String] = run { both { b ⇒ drunkFlip(b, b) } }
```

The example illustrates how handlers can be composed *horizontally* with mixin composition under the condition that they interpret the effects into the same effect domain.

Operationally, they share the same prompt. By subtyping, the combined handler can be used to handle both effects. In `res3`, it is passed down twice, once for each effect it handles.

Safe Forwarding of Effects. Effect handlers allow us to locally handle a subset of effects used by a program. One interesting consequence is that handlers again can use effects in their implementation which are then handled by other handlers. That is, we can compose handlers *vertically* by forwarding.

When compared to the expression problem literature this forwarding to another handler is remindful of family self references (Oliveira *et al.*, 2013) or base algebras (Hofer *et al.*, 2008). We illustrate this with a handler for the parser effect (Leijen, 2016).

```
trait Parser extends Eff {
  def alternative[A, E](fst: A / E, snd: A / E): A / (effect & E)
  def accept(token: Char): Unit / effect
}
```

We can use the parser effect to describe a grammar that counts the number of consecutive occurrences of the letter `'a'` ending in a single letter `'b'`:

```
// AB ::= a AB | b
def AB(p: Parser): Int / p.effect = p.alternative(
  for { _ ← p.accept('a'); rest ← AB(p) } yield rest + 1,
  for { _ ← p.accept('b') } yield 0)
```

To implement the parser effect, let us assume an effect signature and a handler implementation for reading characters from an input stream.

```
trait Input extends Eff { def read(): Char / effect }
def reader[R, E](s: String)(prog: (in: Input) ⇒ R / (in.effect & E)): R / E
```

Equipped with handlers for nondeterministic choice, exceptions and reader we can implement a handler for `Parser`.

```
trait ParserForward(
  val exc: Exc, val amb: Amb, val in: Input
) extends Parser {
  type effect = exc.effect & amb.effect & in.effect
  def accept(expected: Char): Unit / effect = for {
    next ← in.read()
    res ← if (next == expected) pure(()) else exc.raise()
  } yield res
  def alternative[A, E](fst: A / E, snd: A / E) =
    amb.flip() flatMap { b ⇒ if (b) fst else snd }
}
```

By expecting instances for `Exc`, `Amb` and `Input` as constructor arguments, the `ParserForward` trait makes explicit that it depends on three capabilities. It closes over the handler instances which in turn might close over prompts. Previously, we expressed that we use the effect of capturing the continuation delimited by `prompt`, by defining `type effect = prompt.type`. Similarly, we now define `effect` as the intersection of the effect types we forward to. This

ties the scope of the parser capability to the intersection of scopes of the used capabilities `exc`, `amb` and `in` and thereby guarantees effect safety.

Reusing the handler functions `both` and `reader` we can finally define the handler function for parsers

```
def parse[R](lang: (p: Parser) ⇒ R / p.effect)(s: String) =
  both { b ⇒ reader(s) { in ⇒ lang(new ParserForward(b, b, in) {}) } }
```

and use it to parse example strings:

```
run { parse(AB)("b") }   //> Some(0)
run { parse(AB)("aab") } //> Some(2)
run { parse(AB)("xab") } //> None
```

5.3. Effect Handlers and Object Orientation

Effekt is an embedding of effect handlers in a language with support for object oriented programming. Naturally the question arises how these two features interact.
Object oriented programming has a strong focus on encapsulation. In particular, the concrete implementation of an object and its internal state is often hidden behind an interface. That is, the implementation can differ with the granularity of a single object. Another important feature is that objects are first-class and typically must be stored on the heap.

In contrast, effects and handlers are tied to a stack discipline. Effect handlers can capture parts of the stack as a continuation, prompts delimit segments of the stack and effect typing asserts that these stack operations are safe.

Which effects are used by an object's implementation can either be seen as part of the public interface or as a private implementation detail. It is a design decision the programmer should make. However, if the effects used by an object are hidden behind an interface, how can we assert effect safety? For instance, if an object closes over a capability, the object's lifetime needs to be restricted to the capability's lifetime. Otherwise the use of the capability might not be effect safe.

In this section, we will discuss possible design choices when combining effect handlers with object oriented programming while maintaining effect safety. The following interface will serve as a running example:

```
trait Person {
  def greet(other: String): Unit
}
```

5.3.1. Alternative 1. Effects as Part of the Public Interface

An implementation of this interface might want to use the following effect to print the greeting on the console.

```
trait Console extends Eff { def print(msg: String): Unit / effect }
```

However, the method `greet` as defined above does not mention the `Console` effect. Of course, we can change the interface accordingly.

```
trait Person {
  def greet(other: String)(out: Console): Unit / out.effect
}
```

Now, the `Console` effect is part of the public interface and all implementations of `Person` can make use of it to implement method `greet`. The effect has to be handled by the caller of `greet`. In this variant, it is possible to have multiple implementations of `Person` and store the instances in data structures on the heap.

```
var p1: Person = new Person { ... }
var p2: Person = new Person { ... }
val ps = List(p1, p2)
```

5.3.2. Alternative 2. Hiding Effects behind an Interface

Changing the interface of `Person` to mention the effects used by a particular implementation leaks implementation details that we might want to encapsulate. As in the previous sections, we can hide the effects behind an abstract type member `effect`.

```
trait Person {
  type effect
  def greet(other: String): Unit / effect
}
```

Just like a handler closes over a prompt, an implementation of `Person` would close over the effect capabilities:

```
class MyPerson(val out: Console) extends Person {
  type effect = out.effect
  def greet(other: String) = out.print("Hello " + other)
}
```

This way the lifetime of an object of type `MyPerson` is coupled to the lifetime of the capability `out`.

```
withConsole { out ⇒
  ...
  val p = new MyPerson(out)
  ...
}
```

For instance, the object `p` must not leave the scope of `withConsole` which is ensured by our effect system: `out.effect` is an abstract type that only unifies with this one particular call to `withConsole`.

To be eventually able to handle the effects used by the implementation, users thus always need to have stable paths to an object.

```
def user(p1: Person, p2: Person): Unit / (p1.effect & p2.effect) = for {
  _ ← p1.greet("Alice")
  _ ← p2.greet("Bob")
} yield ()
```

In this example p_1 and p_2 are arguments of method `user` and thus have stable paths that can be used path-dependently in the return type. In general, the requirement of path stability excludes objects to be stored in mutable references or in containers like lists. While we can store p_1 in a mutable variable, the effect system will prevent us from calling any effectful methods on it.

5.3.3. Alternative 3. Grouping Objects by their Effect Implementations

The first alternative requires all objects to use the same effects in their implementation and the second alternative allows each object to individually differ in their effect implementation. Both solutions also have drawbacks: the former constrains the implementer while the latter imposes restrictions on the user.

As a compromise between the two extremes, we can generalize over the effect implementation and thereby group objects by their effect implementations.

```
trait Person[E] {
  def greet(other: String): Unit / E
}
```

Like with abstract type members, implementing classes can instantiate `E` to the desired implementation effects. Like with the first alternative, objects of type `Person[Console]` leak the implementation detail that they use the `Console` effect in their implementation.

Users can be polymorphic in the effect type:

```
def user[E](p1: Person[E], p2: Person[E]): Unit / E = for {
  _ ← p1.greet("Alice")
  _ ← p2.greet("Bob")
} yield ()
```

While we now can store objects of type `Person[E]` in mutable references or lists of type `List[Person[E]]`, this requires all instances to have the same effect implementation.

6. Related Work

Our implementation of delimited control is based on Dybvig *et al.* (2007). While Dybvig *et al.*, aim to be as general as possible, we specialize their library for control operators with first-class prompts to only expose `reset` and $\texttt{shift}_0$. In order to achieve effect parametricity we have to make sure that each prompt uniquely determines a `reset`. Therefore in Effekt every `reset` introduces a *fresh* prompt. Being based on $\texttt{shift}_0$ and introducing a fresh prompt per `reset`, our control operators are closely related to `spawn` / `controller` (Hieb & Dybvig, 1990) as illustrated by the following equation:

```
def spawn(body) = reset { p ⇒ body(shift(p)) }
```

Like our presentation in Section 3, Dybvig *et al.*, guarantee answer type safety by indexing prompts with the expected answer type. Furthermore, they use rank-2 types to prevent prompts from being used across different instances of `run`. But, as they observe, this is not enough to achieve effect safety, which they explicitly leave to future work. We solve this problem in Section 4. We use abstract type members instead of rank-2 types to achieve better type inference in Scala.

Kiselyov & Shan (2008) generalize resource safety from a single region to multiple nested regions. They achieve region polymorphism and region subtyping together with good type inference for their library in Haskell. On the type level they represent nested regions as multiple applications of a monad transformer while we represent nested delimiters by an intersection of singleton prompt types. To achieve region polymorphism they reuse Haskell's polymorphism and to achieve region subtyping they use Haskell's type class instance search. To achieve effect polymorphism we reuse Scala's polymorphism and to achieve subtyping we reuse subtyping for intersection types built into Scala.

Most languages with effect handlers base their effect system on some form of row polymorphism. Prominent examples are Koka (Leijen, 2014), Frank (Lindley *et al.*, 2017), and Links (Hillerström & Lindley, 2016; Hillerström *et al.*, 2017). In contrast, effect safe library embeddings like Extensible Effects (Kiselyov *et al.*, 2013; Kiselyov & Ishii, 2015) use various forms of open union types to track the list of unhandled effects. In Effekt, we index the monad for delimited control with an intersection of all effects used by a computation. Parreaux *et al.* (2018) apply a very similar strategy to implement hygienic macros. The type parameter `Ctx` of the type `Code[+Typ, -Ctx]` is used to track the set of free variables:

```
class Variable[A] {
  type Ctx;
  def substitute[T,C](pgrm: Code[T, Ctx & C], v: Code[A, C]): Code[T,C]
}
```

As can be seen from the type of `substitute`, substitution of free variables removes `Ctx` from the intersection type and thus corresponds to handling of effects.

Both, the dynamic and static semantics of Effekt is closely related to $\lambda_{\Downarrow\Uparrow}$ presented by Zhang & Myers (2019). As in previous versions of Effekt (Brachthäuser & Schuster, 2017), capabilities in $\lambda_{\Downarrow\Uparrow}$ are tuples of a label and the handler implementation. Also like in Effekt, they are explicitly passed to the use-site of the effect. Handling an effect introduces a fresh label. Like prompts in Effekt, the label is used on the term level to delimit the scope of captured continuations. Like the singleton type of prompts in the present paper, the label is also used on the type level to track the set of unhandled effects. Due to the embedding of Effekt in Scala, prompts are first class while labels in $\lambda_{\Downarrow\Uparrow}$ are not first class. Instead, the binding of a label by means of `try`, and the use of a label in a handler implementation is statically scoped.

The effect system as presented in this paper is heavily influenced by the one of $\lambda_{\Downarrow\Uparrow}$. To ensure effect safety, Zhang & Myers use a simple form of dependent types: Using an effect handler `h` introduces `h.lbl` in the effect row which is effectively a set of labels. This dependent effect type can only be discharged by the very same reset that binds the label. Zhang & Myers formalize $\lambda_{\Downarrow\Uparrow}$ and formally show effect parametricity. However, they do not provide an implementation of their calculus. We use intersection types and path dependent types to encode the ideas of the $\lambda_{\Downarrow\Uparrow}$ effect system and thereby make Effekt effect safe.

In earlier work on Effekt (2017; 2018), we started to explore the combination of effect handlers and object orientation. However, those versions of Effekt did not guarantee effect safety. The present paper shows how to add effect safety to Effekt, support effect polymor-

phism, and effect parametricity. Indexing the monad for delimited control with the set of used effects is essential to guarantee effect safety. It is not immediate to us how the effect system can be embedded in a direct style version of the library (Brachthäuser *et al.*, 2018) because there is no monadic type that could carry the set of used effects. In earlier versions of Scala Effekt, `handle` did not create a fresh prompt, but used the given handler instance as a prompt. But this would allow for accidental handling of operations, hence we changed the interface of the library.

As highlighted in Section 5, effect safe programming with effect handlers in a language with objects comes with new challenges – mediating encapsulation and flexible use of objects. Inostroza & van der Storm (2018) also combine effect handlers and object orientation in the language JEff. In JEff, the continuation takes an updated copy of the effect handler as additional argument. This allows both to model dynamically scoped state (Kiselyov *et al.*, 2006) and to change the handler implementation for the rest of the computation, similar to shallow handlers. The effect system of JEff does not feature effect polymorphism and hence problems with effect encapsulation do not arise.

7. Conclusion

In this paper, we presented Effekt, a monadic library for programming with effect handlers in Scala that features effect polymorphism, effect subtyping and effect safety. We use intersection types and path-dependent types to track the set of effects a program might use. This allowed us to directly reuse Scala's support for polymorphism for effect polymorphism and Scala's support for subtyping for effect subtyping. Combining effect handlers with object oriented programming both offers new ways to modularize effectful programs but also comes with new challenges.

References

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1), 108–123.

Biernacki, Dariusz, Piróg, Maciej, Polesiuk, Piotr, & Sieczkowski, Filip. (2018). Handle with care: Relational interpretation of algebraic effects and handlers. *Proceedings of the Symposium on Principles of Programming Languages*. ACM.

Brachthäuser, Jonathan Immanuel, & Schuster, Philipp. (2017). Effekt: Extensible algebraic effects in scala (short paper). *Proceedings of the International Symposium on Scala*. ACM.

Brachthäuser, Jonathan Immanuel, Schuster, Philipp, & Ostermann, Klaus. (2018). Effect handlers for the masses. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM.

Carette, Jacques, Kiselyov, Oleg, & Shan, Chung-Chieh. (2007). Finally tagless, partially evaluated. *Pages 222–238 of: Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer LNCS 4807.

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Proceedings of the Conference on LISP and Functional Programming*. ACM.

Dolan, Stephen, White, Leo, & Madhavapeddy, Anil. (2014). Multicore ocaml. *OCaml Workshop*, vol. 2.

Dolan, Stephen, Eliopoulos, Spiros, Hillerström, Daniel, Madhavapeddy, Anil, Sivaramakrishnan, KC, & White, Leo. (2017). Concurrent system programming with effect handlers. *Proceedings of the Symposium on Trends in Functional Programming*.

Dybvig, R Kent, Jones, Simon Peyton, & Sabry, Amr. (2007). A monadic framework for delimited continuations. *Journal of functional programming*, 17(6), 687–730.

Forster, Yannick, Kammar, Ohad, Lindley, Sam, & Pretnar, Matija. (2017). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proceedings of the International Conference on Functional Programming*. ACM.

Gaster, Benedict R, & Jones, Mark P. (1996). A polymorphic type system for extensible records and variants. *Technical Report NOTTCS-TR-96-3 – Department of Computer Science, University of Notthingham*.

Haller, Philipp, & Loiko, Alex. (2016). Lacasa: Lightweight affinity and object capabilities in scala. *Pages 272–291 of: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM.

Hieb, R., & Dybvig, R. Kent. (1990). Continuations and concurrency. *Pages 128–136 of: Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. PPOPP '90. ACM.

Hillerström, Daniel, & Lindley, Sam. (2016). Liberating effects with rows and handlers. *Proceedings of the Workshop on Type-Driven Development*. New York, NY, USA: ACM.

Hillerström, Daniel, Lindley, Sam, Atkey, Bob, & Sivaramakrishnan, KC. (2017). Continuation passing style for effect handlers. *Formal Structures for Computation and Deduction*. LIPIcs, vol. 84. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Hofer, Christian, Ostermann, Klaus, Rendel, Tillmann, & Moors, Adriaan. (2008). Polymorphic embedding of DSLs. *Proceedings of the Conference on Generative Programming and Component Engineering*. ACM.

Hudak, Paul. (1998). Modular domain specific languages and tools. *Pages 134–142 of: Proceedings of the Conference on Software Reuse*. IEEE Computer Society Press.

Inostroza, Pablo, & van der Storm, Tijs. (2018). Jeff: Objects for effect. *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. New York, NY, USA: ACM.

Kammar, Ohad, Lindley, Sam, & Oury, Nicolas. (2013). Handlers in action. *Pages 145–158 of: Proceedings of the International Conference on Functional Programming*. ACM.

Kennedy, Andrew. (2007). Compiling with continuations, continued. *Pages 177–190 of: Proceedings of the International Conference on Functional Programming*. ACM.

Kiselyov, Oleg, & Ishii, Hiromi. (2015). Freer monads, more extensible effects. *Pages 94–105 of: Proceedings of the Haskell Symposium*. ACM.

Kiselyov, Oleg, & Shan, Chung-chieh. (2008). Lightweight monadic regions. *Proceedings of the Haskell Symposium*. Haskell '08. ACM.

Kiselyov, Oleg, & Sivaramakrishnan, KC. (2016). Eff directly in ocaml. *ML Workshop*.

Kiselyov, Oleg, Shan, Chung-chieh, & Sabry, Amr. (2006). Delimited dynamic binding. *Pages 26–37 of: Proceedings of the International Conference on Functional Programming*. New York, NY, USA: ACM.

Kiselyov, Oleg, Sabry, Amr, & Swords, Cameron. (2013). Extensible effects: An alternative to monad transformers. *Pages 59–70 of: Proceedings of the Haskell Symposium*. ACM.

Landin, Peter J. (1965). A generalization of jumps and labels. *Report, UNIVAC Systems Programming Research*.

Launchbury, John, & Sabry, Amr. (1997). Monadic state: Axiomatization and type safety. *Proceedings of the International Conference on Functional Programming*. ICFP '97. ACM.

Leijen, Daan. (2014). Koka: Programming with row polymorphic effect types. *Proceedings of the Workshop on Mathematically Structured Functional Programming*.

Leijen, Daan. (2016). *Algebraic effects for functional programming*. Tech. rept. MSR-TR-2016-29. Microsoft Research technical report.

Leijen, Daan. (2017a). Structured asynchrony with algebraic effects. *Pages 16–29 of: Proceedings of the Workshop on Type-Driven Development*. ACM.

Leijen, Daan. (2017b). Type directed compilation of row-typed algebraic effects. *Pages 486–499 of: Proceedings of the Symposium on Principles of Programming Languages*.

Leijen, Daan. (2018). First class dynamic effect handlers: Or, polymorphic heaps with dynamic effect handlers. *Proceedings of the Workshop on Type-Driven Development*. ACM.

Lindley, Sam. (2018). Encapsulating effects. *Algebraic Effect Handlers go Mainstream (Dagstuhl Seminar 18172)*, vol. 8. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Lindley, Sam, McBride, Conor, & McLaughlin, Craig. (2017). Do be do be do. *Pages 500–514 of: Proceedings of the Symposium on Principles of Programming Languages*. ACM.

Odersky, Martin, & Zenger, Matthias. (2005a). Independently extensible solutions to the expression problem. *Proceedings of the Workshop on Foundations of Object-Oriented Languages*.

Odersky, Martin, & Zenger, Matthias. (2005b). Scalable component abstractions. *Pages 41–57 of: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM.

Oliveira, Bruno C. d. S., & Cook, William R. (2012). Extensibility for the masses: Practical extensibility with object algebras. *Pages 2–27 of: Proceedings of the European Conference on Object-Oriented Programming*. Springer LNCS 7313.

Oliveira, Bruno C. d. S., van der Storm, Tijs, Loh, Alex, & Cook, William R. (2013). Feature-oriented programming with object algebras. *Proceedings of the European Conference on Object-Oriented Programming*. Springer LNCS 7920.

Parreaux, Lionel, Voizard, Antoine, Shaikhha, Amir, & Koch, Christoph E. (2018). Unifying analytic and statically-typed quasiquotes. ACM.

Plotkin, Gordon, & Pretnar, Matija. (2009). Handlers of algebraic effects. *Pages 80–94 of: European Symposium on Programming*. Springer-Verlag.

Schuster, Philipp, & Brachthäuser, Jonathan Immanuel. (2018). Typing, representing, and abstracting control. *Proceedings of the Workshop on Type-Driven Development*. ACM.

Swierstra, Wouter. (2008). Data types à la carte. *Journal of functional programming*, 18(04), 423–436.

Wadler, Philip. 1998 (Nov.). *The expression problem*. Note to Java Genericity mailing list.

Wu, Nicolas, Schrijvers, Tom, & Hinze, Ralf. (2014). Effect handlers in scope. *Proceedings of the Haskell Symposium*. Haskell '14. ACM.

Zhang, Yizhou, & Myers, Andrew C. (2019). Abstraction-safe effect handlers via tunneling. ACM.